THE DEVELOPMENT OF mod_antiCrawl: AN ANTI CRAWLER ADD-ON MODULE FOR APACHE WEB SERVERS

by

Muhammed Oğuzhan Topgül

B.S, Computer Engineering & Electronics Engineering,

Kadir Has University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2012

ACKNOWLEDGEMENTS

I sincerely thank to my thesis supervisor Prof. Ufuk Çağlayan for his support and encouragement throughout my master education and thesis. I'm very grateful for his guidance, patience and understanding.

I am very thankful to TÜBİTAK-BİDEB for supporting me financially in my MS education.

I would like to thank to my managers Tahsin Türkoz and Yılmaz Çankaya for their patience and support; my colleagues Can Bican, Gökhan Alkan, Necati Şişeci, Bakır Emre and my friend Bünyamin Demir for their guidance and help in technical perspective.

Also, I would like to thank to my family for their love and moral support during all difficult times throughout my study and life.

Finally, I would like to thank to my wife, for her love, patience, encouragement and support in every steps of my life since the day I met her.

ABSTRACT

THE DEVELOPMENT OF mod_antiCrawl: AN ANTI CRAWLER ADD-ON MODULE FOR APACHE WEB SERVERS

A web crawler can be defined as automated software that extracts website maps by visiting all the links in a website. Website map extraction process can be used to build a basis for a web attack. Hence, crawling plays an important role in automated attacks. The most automated vulnerability scanners perform crawling before vulnerability tests in order to determine overall map and attack surface. Besides automated scanning features, crawlers can also be used for content theft. By utilising a crawler, one can copy all the pages and content of a website by visiting all pages in an orderly manner. Anti-crawling can be defined as a set of mechanisms that prevents websites from being crawled by automated crawlers. In this thesis, a set of anti-crawling mechanisms are combined into an Apache web server module called mod_antiCrawl. mod_antiCrawl is developed in C language by using Apache API and it has crawler detection and inhibition capabilities to protect servers from malicious crawlers. The performance of mod_antiCrawl has also been studied and our results show that website map discovery by crawlers decreases at least 70% after mod_antiCrawl is activated. This ratio increases to 90% by enabling different functionalities of the module.

ÖZET

mod_antiCrawl: APACHE WEB SUNUCUSU İÇİN İNTERNET ROBOTU ENGELLEYİCİ (ANTI CRAWLER) EKLENTİ MODÜLÜ GELİŞTİRİLMESİ

İngilizcede crawler diye adlandırılan İnternet Robotu yazılımları bir web sayfasındaki tüm bağlantıları gezerek bu sitenin haritasını çıkartan otomatikleştirilmiş yazılımlardır. Bir web sitesinin haritasının çıkartılması, o siteye yapılacak bir saldırı için temel teşkil edeceğinden otomatikleştirilmiş saldırılar için büyük önem taşımaktadır. Bu yüzdendir ki otomatik web açıklık tarayıcılarının hepsi, taramaya başlamadan önce mutlaka sitenin haritasını çıkartmak için bağlantı keşfi işlemi (crawling) gerçekleştirir. İnternet robotları, otomatik web açıklık taramalarına temel olmak dışında, içerik hırsızlığı için de sıkça kullanılmaktadır. Otomatikleştirilmiş bir şekilde bir sitenin tüm içeriğinin sayfa sayfa gezilerek başka bir web sitesine kopyalanması konusunda internet robotları büyük rol oynamaktadır. Bu tezde web sunucular için internet robotlarına karşı bağlantı keşfi önleyici yöntemleri içeren internet robotu engelleyici modülü geliştirilmiştir. mod_antiCrawl C dili ve Apache API'si kullanılarak yazılmış bir Apache modülüdür. mod_antiCrawl'ın internet robotu yakalama ve engelleme yetenekleri sayesinde sunucular, zararlı robot yazılımlardan korunmaktadır. Yapılan performans değerlendirme ölçümlerinde, modül aktif konuma getirildikten sonra internet robotlarının elde edebildikleri bulgu sayısında en az %70'lik bir düşüş sağlandığı görülmüştür. Bu oran mod_antiCrawl içerisindeki internet robotı karakteristiğine daha uygun fonskiyonların da yardımıyla %90 seviyesine çıkabilmektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF ACRONYMS/ABBREVIATIONS x
1. INTRODUCTION 1
2. WEB CRAWLERS AND ANTI-CRAWLING MECHANISMS 4
2.1. Details of Web Crawlers 4
2.1.1. Frontier 5
2.1.2. Crawl History 6
2.1.3. Fetching 6
2.1.4. Parsing 6
2.2. Multi-threaded Crawlers 8
2.3. Anti-Crawling Mechanisms 9
2.3.1. Cloaking 9
2.3.2. Positive User-Agent Lists 10
2.3.3. Session ID Check 10
2.3.4. Hidden Links 10
2.3.5. Using JavaScript 10
2.3.6. Spider Traps 11
2.3.7. Request Density Control 11
3. DEVELOPMENT 12
3.1. Models for mod_antiCrawl 12
3.1.1 Request Density Model 13
3.1.2. Hidden Link Model 14
3.2. mod_antiCrawl Design and Implementation 17
3.2.1. Hash Tables 21
3.2.2. Mutexes 22
4. PERFORMANCE EVALUATION 24

4.1. Performance Measures and Testing Environment 24
4.2. Crawler Discovery Tests 24
4.3. Server Load Tests 28
5. CONCLUSION AND FUTURE WORK 32
APPENDIX A: mod_antiCrawl USER MANUAL 34
A.1. mod_antiCrawl Installation 34
A.2. mod_antiCrawl Configuration 34
APPENDIX B: mod_antiCrawl DIAGRAMS 37
B.1. Request Density Model UML Activity Diagram 37
B.2. Hidden Link Model UML Activity Diagram 38
B.3. mod_antiCrawl UML Activity Diagram 39
REFERENCES 40
REFERENCES NOT CITED 41

LIST OF FIGURES

Figure 2.1. Basic Steps of Web Crawling Loop. 5

Figure 2.2. Parts of Parsing Process. 6

Figure 2.3. An Example Tag Tree. 8

Figure 2.4. Thread of Multi-threaded Crawler. 9

Figure 3.1. Example Code Before Hidden URL Injection. 16

Figure 3.2. Example Code After Hidden URL Injection. 16

Figure 3.3. Example of Choosing Inappropriate Keyword For Hidden URL Injection. 17

Figure 3.4. Overall Structure of Apache Server and mod_antiCrawl. 18

Figure 3.5. Pseudo Code of the Access Checking Procedure of mod_antiCrawl. 20

Figure 3.6. Example Hash Table With Separate Chaining [10]. 21

Figure 3.7. The View of the Blacklist and the Timetable in a Hash Table. 22

Figure 4.1. The Test Setup of mod_antiCrawl Performance Evaluations. 24

Figure 4.2. The Table of the Number of Files Found in Different Configurations. 26

Figure 4.3. The Graph of the Number of Files Found in Different Configurations. 27

Figure 4.4. Response Time of The Apache Server When mod_antiCrawl is Inactive. 29

Figure 4.5. Response Time of The Apache Server When mod_antiCrawl is Active. .. 30

Figure 4.6. Free Memory of the Server When mod_antiCrawl is Deactivated. 30

Figure 4.7. CPU Load of the Server When mod_antiCrawl is Deactivated. 30

Figure 4.8. Free Memory of the Server When mod_antiCrawl is Activated. 31

Figure 4.9. CPU Load of the Server When mod_antiCrawl is Activated. 31

Figure A.1. Apache Module Configuration Example. 35

Figure A.2. Apache Module Configuration. 36

Figure B.1. RDM UML Activity Diagram. 37

Figure B.2. HLM UML Activity Diagram. 38

Figure B.3. mod_antiCrawl UML Activity Diagram. 39

LIST OF ACRONYMS/ABBREVIATIONS

apxs	Apache Extensions Tool
API	Application Programming Interface
CPU	Central Processing Unit
DOM	Document Object Model
DOS	Denial Of Service
DDOS	Distributed Denial Of Service
GB	Gigabyte
GHz	Gigahertz
HLM	Hidden Link Model
href	Destination Address Of A Link In HTML
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IP	Internet Protocol
I/O	Input-Output
KB	Kilobyte
MUTEX	Mutual Exclusion
PHP	Hypertext Preprocessor
RDM	Request Density Model
SEO	Search Engine Optimization
SNMP	Simple Network Managing Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator

1. INTRODUCTION

Web crawling is the process of browsing web page links in an automated way and web crawler is the program that is used to create a map or link tree of a website by visiting web page links automatically. The map is generally used for indexing any website or copying contents of a web or producing an attack basis for automated tools. An automated web scanner crawls the website as the first step and then applies attack patterns on web pages that were previously crawled.

A web scrapper, which is also called a web harvester is a tool that is used for extracting meaningful data from websites. Web scrapping is closely related to web indexing, which is a universal technique adopted by most search engines to index web content by using automated bots. In contrast, web scrapping focuses mainly on the transforming unstructured web content.

Hot linking, which is also called inline linking is the use of linked objects into a website that belong to another website. Hot linking is generally used for linking images from other websites. By hot linking, hit count of the owner website does not increase but the bandwidth of it becomes used.

Crawling and indexing techniques are generally used by search engines to provide faster services and more precise results. On the other hand, attackers are using such techniques for building strong attack basis. All the automated vulnerability scanners perform crawling before applying attack patterns to a target website. Attackers generally use automated tools to carry out attacks more easily. Thus, it can be said that if websites have successful anti-crawling mechanisms, then they could not be scanned by automated vulnerability scanners.

Another abuse of web crawlers is about content theft. By utilizing a crawler, one can copy all the pages and content of a website by visiting all pages in an orderly manner. Thereby, imitation website gets hits through clicks and gains profit without any labor. Such

imitation websites may also be used for social engineering attacks as well as phishing attacks. Additionally, abusers also use inline linking method to link images and use target websites' bandwidth without increasing the hit count of it.

In web application attacks, it is crucial to determine the general view, or general perspective of the website, which is called the URL map. URL map extraction is a part of so called "discovery" phase, which attacker tries to identify assets and attack vectors in order to penetrate the system. Generally discovery phase takes more time in comparison to other phases. If one wants to perform an effective attack, the first thing that can be done is decreasing discovery time consumption. Hence, towards achieving fast and efficient attack, automated tools are widely used.

In terms of discovery phase, web crawlers are the most important automated tools of a web attack. Because all the automated web vulnerability scanners use crawlers in order to extract URLs and attack points in a website. If the crawlers can be denied, automated vulnerability scanning process gets damage and loses its effectiveness and accuracy.

Moreover, web crawlers are also used for content theft and content copying. Crawlers visit all the pages of a website and copy the content of these pages into another. By the help of web crawlers, significant amount of resources can be analyzed, parsed and extracted in an uncontrolled way without any permission.

There are some protections for Denial of Service (DOS) and Distributed Denial of Service (DDOS) attacks that can be used as a crawler guard particularly. However, they are not complete because of the differences in characteristics of DDOS tools and crawlers. Since attackers do not wait for any responses in DDOS attacks, they can use spoofed IPs - multiple fake source IPs- in order to send requests. Denial of service attacks are generally done by just sending a huge number of requests. Hence, it does not need to wait for any response. But in web crawling process the request-response mechanism must be used, because crawler uses HTTP responses in order to decide if the web page or link is valid or not. Therefore the prevention technique has to be designed according to request-response mechanism.

Apache is a web server which is mainly developed and maintained by open community that is supported by Apache Software Foundation and licensed under Apache License [1]. According to [2], Apache is the most popular web server since 1996 and it has currently 65% market share around the world. Due to the multiplatform compliance and community behind it, there are more than 500 add-on modules [3]. Thus, in this study it is decided to develop an Apache module for the benefit of community.

In this thesis, a set of anti-crawling mechanisms are combined into an Apache web server module called mod_antiCrawl. mod_antiCrawl is developed in C language by using Apache API and it has crawler detection and inhibition capabilities to protect servers from malicious crawlers. The performance of mod_antiCrawl has also been studied and our results show that website map discovery by crawlers decreases at least 70% after mod_antiCrawl is activated. This ratio increases to 90% by enabling different functionalities of the module.

This thesis is organized as follows: In Chapter 2, crawling and anti-crawling terms are defined; details of crawling process and different kinds of crawler protections are explained. In Chapter 3, design details and abilities of developed module are presented. In Chapter 4, detection performance, advantages and disadvantages of developed module are evaluated. Chapter 5 concludes the overall work.

2. WEB CRAWLERS AND ANTI-CRAWLING MECHANISMS

2.1. Details of Web Crawlers

Web crawler is the tool that downloads entire website or a part of it by visiting hyperlinks in the web pages. Crawler starts from the page called seed and by using hyperlinks on the seed page, it finds other web pages. After that, it uses these new web pages as seed pages and extracts some new pages from them. Crawling process is an iterative process and crawlers follow these steps until they reach the objective of the crawl. The objective might be to download entire website, or visit all URLs in it.

Web has a dynamic content and it grows every moment of time. Hence, doing a crawling process for once is never enough. Since crawling results might become useless in a very short period of time, it is very important to crawl web pages in a recursively manner and get very fresh and updated content.

Search engines are generally use web crawlers in order to decrease the searching effort by searching and indexing web pages previously. Because of the blind and comprehensive behavior of these search engine crawlers, they called exhaustive crawlers. Some other crawlers behave selective in their web page choices. These crawlers called topical crawlers or focused crawlers due to the fact that their scans are topic based.

Web crawler first creates the frontier -a list of unvisited URLs- and picks and crawls the URLs in a sequence. It parses the crawled page, and extracts new URLs and application specific information from that page. At the end of the process, crawler adds extracted new unvisited URLs into frontier again [4]. Loop continues until the stopping criteria is reached.

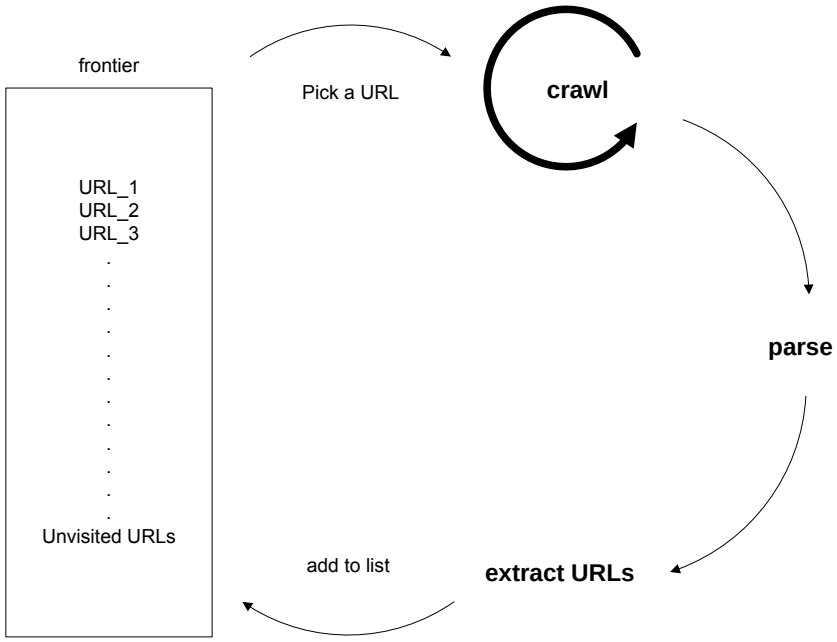

Figure 2.1. Basic Steps of Web Crawling Loop.

Crawling process can be seen as a graph search problem. In this approach, nodes are web pages and edges are hyperlinks. Crawler starts from nodes and follow edges to reach other nodes. Link extraction in crawling can be seen as edge extraction in graph search.

2.1.1. Frontier

Frontier holds the links that is going to be visited like a to-do list. However it has a size limit. Therefore, same link should not be placed into frontier more than once. In order to prevent this kind of duplication, it must be checked if there is a record in frontier for the same URL.

If the frontier was designed as a priority queue, the URL at the top of the list is the URL that has highest score. Crawling starts from the highest scored URL and extracted URLs are scored according to an algorithm in order to determine the priority of the URL in the list.

2.1.2. Crawl History

Crawl history is the file that crawled URLs are hold with their timestamp values. It shows the crawling path from the first URL in the frontier. Crawl history is used for prevention to visit previously visited URLs as well. Visited URLs can be stored simply as files, but in this case filenames must be unique. Because it is not allowed to use same name as file name in file system. As a solution, using hash values of the URLs as filenames is effective.

2.1.3. Fetching

In fetching process it is very important to parse the HTTP response headers for status codes and redirections. “HTTP 200” header means the requested web page is valid and there is no problem in server side or client side. That’s why the description of the HTTP 200 status code is “200 OK!” If client gets HTTP 404 code, that means the requested web page cannot be found on server and there is nothing to fetch. Another important point of fetching is the freshness and the age of the page or document. The “last-modified” header of the HTTP response is used to determine the age of the document. Because, it says the last modification time of the page or document.

2.1.4. Parsing

Once a web page has been fetched, the next step of the crawling process is parsing the fetched web page’s content and using these parsed values as a future crawl path. There are two parts of parsing process: simple hyperlink extraction and HTML tag tree analyze.

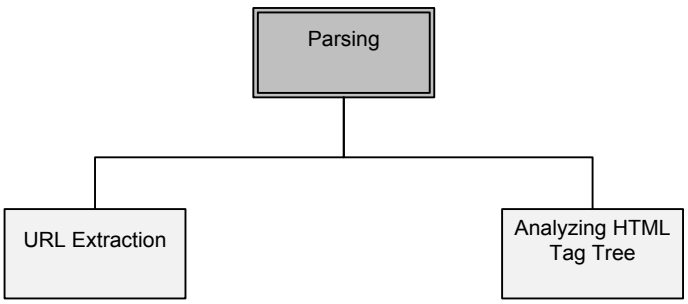

Figure 2.2. Parts of Parsing Process.

2.1.4.1. URL Extraction and Canonicalization. HTML parsers mainly provide the functionality of detecting and identifying HTML tags and associated attribute values of these tags. While extracting hyperlinks, parsers first identify HTML <*a*> tags –that's called anchor tags- in the source code and then parse the value of the *href* attribute of the tag.

In order to avoid fetching the same page for many times, all the different URLs that correspond to a single web page can be mapped onto a single canonical form. Some of the canonicalization process can be shown as;

- Convert all URL including protocol name into lowercase.
- Remove 'anchor' part of the URL. (Anchor part is the followings of # character)
- Perform URL encoding.
- Remove trailing forward slash "/" character.
- Remove the tailing default web pages like index.html in order to retrieve base URL.
- Add port 80 when no port number is specified. Because port number 80 is the default HTTP port [5].

2.1.4.2. HTML Tag Tree. Since URLs are generally put into the frontier according to their priorities, HTML tag tree analyze is generally done in order to assess the value of a URL. Hence, crawlers need to utilize tag tree or DOM structure of the HTML pages in order to prioritize web pages by its context. If the aim of the crawler is just extracting URLs, basic HTML parsers should be enough because of the high complexity of HTML DOM tree extraction process.

Crawlers work in a structured manner but many web pages are written in a dirty HTML. For example, in a web page some start tags might not properly ended with an end tag. This kind of badly written HTML is not suitable for structured web crawlers. In order to crawl dirty written web pages, dirty HTML should be corrected by tidying process. Tidying process tries to create an HTML tag tree by putting the missing html tags, but the important point of HTML tidying is each node of the HTML tag tree must be a single parent node.

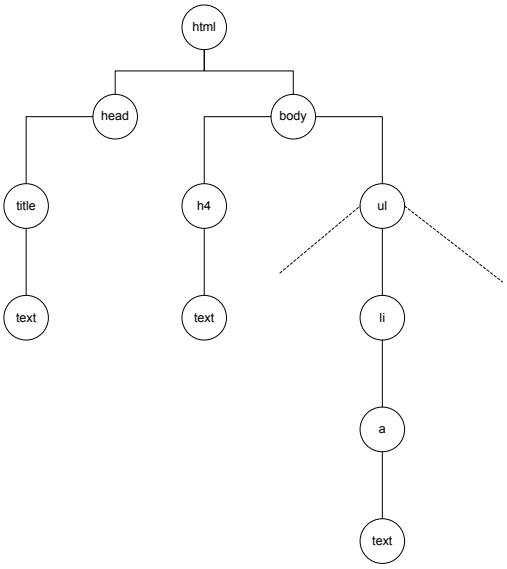

Figure 2.3. An Example Tag Tree.

2.2. Multi-threaded Crawlers

In a sequential crawling loop, time cannot be used efficiently because of the idle times of different components. However, in multi-threaded crawling loop where each thread follows a sequential crawling loop internally, network and process capability is used more efficiently.

In multi-threaded crawling loop, each thread starts with locking up the frontier and gets the URL with the highest priority. After it takes the URL, frontier is being unlocked and other threads can access the frontier. Because of the synchronization of frontier, it should be locked and the access of other threads must be denied while getting a URL. After fetching and parsing processes, the new URLs must be put into frontier even as it is locked to the access of other threads. In multi threaded crawling, the synchronization of the frontier is one of the most important subjects. Synchronization of the crawling history is important as well.

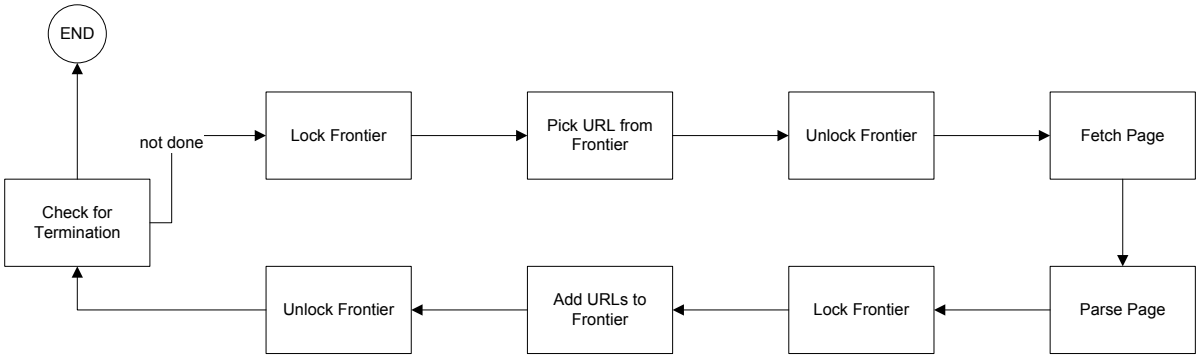

Figure 2.4. Thread of Multi-threaded Crawler.

The multi threaded crawler is generally faces with an empty frontier. In single threaded crawlers, crawling process ends when frontier becomes empty. However, in multi threaded crawling process, empty frontier does not mean the whole crawling process reached to dead end. It is a very high possibility that other threads are fetching web pages and they will add new URLs into frontier in the near future in multi-threaded crawling process. When a thread is faced with an empty frontier, it can be sent to a sleeping state for some time. At the end of that time, the thread wakes up and checks frontier again. These sleeping threads must be tracked by a global monitor and if all the threads are in sleeping state that means the crawler reached to a dead end, and the crawling process is terminated.

2.3. Anti-Crawling Mechanisms

Anti-crawling is the mechanism that prevents websites from being crawled by automated crawlers. Various methods can be used as prevention like cloaking, positive user-agent lists, session-id checks.

2.3.1. Cloaking

Cloaking is a method for deception of the crawlers that mainly depends on the User-Agent header of the HTTP request or the IP address of the user. Cloaking aims to serve different contents for different type of users by differentiating user from its user agent or IP address. Cloaking is actually used for misleading the search engine crawlers to

increase the ranking of the website as a SEO technique. Because of the deception side it is called Black Hat SEO.

2.3.2. Positive User-Agent Lists

User-Agent is an HTTP request header that specifies the software that is used for sending request. Hence, User-Agent parameter can be used to detect web crawlers. Positive User-Agent List contains common and usable user agents like Internet Explorer, Firefox, Safari, and Opera etc. Therefore, if a client sends an HTTP request without a valid User-Agent header value, server denies request or does not reply with an HTTP 200 OK! code because the server assumes this client as a crawler [6].

2.3.3. Session ID Check

Another prevention method is checking session ID of the user, because some crawlers cannot hold session values. If a user does not send a valid session ID in each sequential requests, that means the user is a web crawler. If the crawler has holding session ID capability, more advanced anti-crawling mechanisms should be used [6].

2.3.4. Hidden Links

One of the intelligent methods for crawling prevention is; using invisible hidden links in web pages. Hence regular users browse web pages by web browsers, these links cannot be seen by regular users. On contrary, crawlers follow these invisible links because they generally go over source codes. For example if a client follows this kind of link *<a style=”display:none” href=http://www.example.com/dontfollow.php />* that means, the user cannot be a regular user, it is assumed as a crawler. Because these kinds of invisible links can only be seen by automated tools not by ordinary users [7].

2.3.5. Using JavaScript

Most of the basic, intermediate and some advanced web crawlers do not have mature JavaScript supports. So in order to identify web crawlers, some kind of JavaScript

traps can be used. Because lots of web crawlers handle dynamic content very poorly. An effective JavaScript usage for crawler detection is sensing mouse and keyboard movements. Crawlers are tools that they do not make mouse movements or keyboard strokes; they just parse the source code of website and follow links. So mouse movements and keyboard strokes can be used as a crawler detection method. These detectors can be created with a tiny bit of JavaScript code [7].

2.3.6. Spider Traps

Spider trap is an anti-crawling method that directs crawler to make infinite HTTP requests. As a result it goes into an infinite loop and wastes its resources. As well as ineffective crawling, poorly written crawlers might crash.

2.3.7. Request Density Control

IP and request restriction is one of the basic solutions for crawlers. If so many requests are coming from the same IP address (or session id) in a very short time, that means a crawler is working on the site because normal users can not send that much of requests in a very limited time [8].

3. DEVELOPMENT

Web crawlers are valuable and beneficial tools as long as they are used for increasing search engine performance. However, when they are used for attacking websites or stealing website contents, some precautions called anti-crawling should be taken into account. In this thesis, some major anti-crawling techniques that are presented before are implemented into an Apache module and the designed module is called mod_antiCrawl. Apache modules are plug and play modules. Therefore, they start working when they are activated and stop working when they are deactivated.

In mod_antiCrawl we focus on two main prevention methods.

- If many requests are received from the same IP address in a short period of time, the IP address will be blocked for some time which is specified in the configuration file [8].
- If a client visits the trap link, the client will be caught and blocked for a period of time. This feature is also called Hidden Link Traps [7].

Other prevention methods such as User-Agent control and session id control [6] are not implemented in mod_antiCrawl, because they do not provide efficient protection against web crawlers. Since most of modern crawlers have an option to set User-Agent parameter arbitrarily, User-Agent control becomes easy to bypass. In a similar manner, modern crawlers generally hold session values and send them within every HTTP requests. Therefore, sniffing requests which do not contain a valid session id, is not an effective solution. Additionally, JavaScript traps can be useful when they are implemented in development phase, but they are not applicable for Apache plug and play solutions

3.1. Models for mod_antiCrawl

Detecting automated software is a problem that can simply be expressed to distinguish computer behavior and human behavior. The main purpose of an anti crawler is to identify and distinguish automated computer based requests from human based requests.

Therefore, analyzing computer based and human based request characteristics plays an important role in this study.

One of the most important characteristics of automated HTTP requests is arrival pattern and frequency of the requests, that is also called request density characteristic. Automated crawlers and scanners generally create many requests in a very short period of time, compared to human users. As a result, request density characteristics can be used to differ the crawlers and human users.

Another very important characteristic of the crawler is that a crawler processes the source code of the web page, but human users are interested in the browser view. Hidden objects that cannot be seen by human users in browser view, can be seen by crawlers. In order to identify HTTP requests by crawlers, some hidden objects like hidden links or hidden forms can be used.

In this thesis two of the main characteristics that are stated above are implemented in order to identify web crawler based HTTP requests.

3.1.1. Request Density Model

Request Density Model (RDM) chiefly focuses on the request numbers in a time interval. If large number of requests that cannot be created by human users are received by a server, the server is expected to decline the requests and serve an error page. This is called crawler Punishment in our approach.

In this work, RDM uses two lists for holding IP addresses. One is called the Timetable List which holds the requesting IPs, and the other is called the Blacklist that holds punished IP addresses.

When a request is received by the server, source IP address is extracted and checked whether it is in the Blacklist or not. If the IP is listed in the Blacklist, that means requester client was marked as a crawler and blocked for a period of time before. If the client is in blocking period, Punishment should be continued. In this case, starting time is

updated with the current time and an error page is sent to client. If blocking period is up, the client should be removed from the Blacklist and added into the Timetable, because Punishment is over.

If the requesting IP is not in the Blacklist, it can be in two states;

- This might be the first request from that IP
- The IP might be in the Timetable.

If the IP is listed in the Timetable, the duration between first arrival time and current time should be calculated. If the time difference is higher than a predefined limit given in configuration, the counter of the requesting IP should be reset.

If the time difference is lower than the limit, the value of the counter must be compared with a maximum count value that holds maximum number of requests that can be accepted from same source in a predetermined limited time. When counter exceeds the maximum count value, the IP address is added into the Blacklist and removed from the Timetable list. Because, these requests cannot be originated from a human namely a real user, it might be an automated request. On the other hand, if the counter is below the maximum count value, counter is increased. All of the model stated above can be seen as a UML activity diagram under APPENDIX B.1.

It is important to choose right configurations which are suitable for the system in RDM. Since RDM is a system oriented solution, one configuration that fits to a system might be useless for another. For example, in an application which is receiving hundreds of requests in a second, choosing a configuration like maximum 50 request in 2 seconds makes the website unavailable. Because, such a of configuration will block regular users of the website. Another optimum solution is necessary that is suitable for all kind of systems.

3.1.2. Hidden Link Model

Request number based RDM approach is generally used for different kind of attacks like DOS and DDOS attacks. In the literature, there are some examples of request

number based approaches and these approaches might be useful for crawler protection particularly. However, it is important to understand main qualifications of these attacks in order to produce efficient solutions.

In DOS and DDOS attacks, it is important to send a request to server. The response that is related to the request is not taken into consideration by the attacker. Because of the responseless characteristics of denial of service attacks, spoofed IPs are used in order to send countless number of HTTP requests.

Nevertheless crawlers operate in a different manner. Crawlers have to wait for the response of the request. In other words, crawlers are response dependent tools. Because, crawler decides if the requested page is valid or not according to HTTP response codes and extracts new URLs from response body. Due to the response dependant behavior of crawlers, more effective solutions should be researched.

Hidden Link Model (HLM) approach is designed to find a protection that is suitable for the response dependant behavior. In this approach there are two stages. The first stage is injecting a hidden HTML object into the response and the second stage is trying to trap when the hidden HTML object is requested.

In HLM, it is very crucial to inject an HTML object into every response pages. In this work, injecting an HTML link is preferred because crawlers are visiting URLs in order to find new pages and extract new ones. To perform a hidden URL injection, substitution is chosen as an effective and the least performance consuming method. This operation is done by parsing all the response body and replacing some keywords. Trap URLs are added to keywords as a suffix or prefix. In order to perform a reliable injection, keywords for substitution should be chosen carefully because if the keyword cannot be found in response body, the substitution cannot be done. Hence in this study, keywords are chosen as HTML opening and closing tags like <html>,</html>, <body></body>, <title></title> or <head></head>.

Replacement is performed as follows

- Choose an HTML tag; for example </title>
- Replace </title> with </title>link

"style=display:none;" attribute in <a> tag, to make HTML links invisible to human users. An example injection is performed in source code in Figure 3.1 and Figure 3.2.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
        <title>TITLE OF THE PAGE</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  </head>
  <body>
        <p> HELLO WORLD!!</p>
        <img src="http://www.anticrawl.com/images/banner2.gif">
        <br />
        <a href="index.php"></a>
        <a href="index.php"></a>
        <h1 align="center">HEADING (H1)</h1>
        </body>
</html>
```

Figure 3.1. Example Code Before Hidden URL Injection.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
        <title>TITLE OF THE PAGE</title><a href= a href=hiddenurl.html
style=display:none;>link</a>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  </head>
  <body>
        <p> HELLO WORLD!!</p>
        <img src="http://www.anticrawl.com/images/banner2.gif">
        <br />
        <a href="index.php"></a>
        <a href="index.php"></a>
        <h1 align="center">HEADING (H1)</h1>
        </body>
</html>
```

Figure 3.2. Example Code After Hidden URL Injection.

While injecting a hidden link, opening or closing tag should be chosen by taking web page appearance into account. If the injection is done with <title> opening tag instead of </title> closing tag, the title of the web page becomes damaged as shown in Figure 3.3. The injection should be carried out to protect the appearance and integration of the web page that is served.

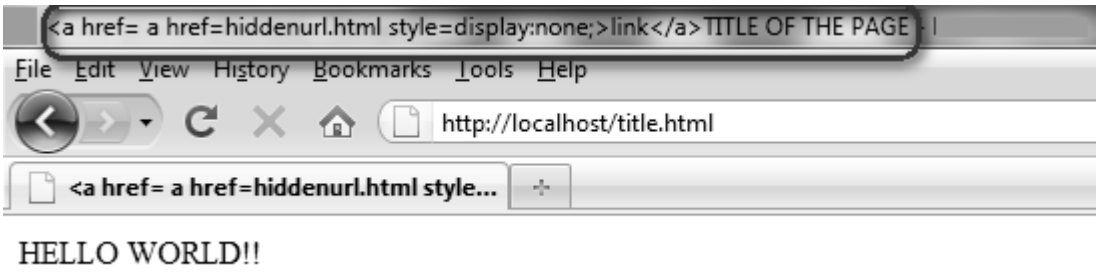

Figure 3.3. Example of Choosing Inappropriate Keyword For Hidden URL Injection.

The next step of injection is to try to catch the injected URL in a HTTP request. When the hidden URL is requested by a client, that means the client is probably an automated tool since the hidden URL cannot be seen by human users. Following links from source code is not a general user behavior; it is automated tool behavior. UML activity diagram of the model is given under APPENDIX B.2.

When a crawler is detected by visiting a hidden trap URL, its IP is added to the Blacklist and blocked for some period of time until the Punishment is up.

3.2. mod_antiCrawl Design and Implementation

Like other web servers, Apache web server is also called Apache httpd is generally used for directing dynamic content to application servers and application servers are responsible from handling these content. Similar to other web servers, Apache can not handle dynamic content by default. But, add-on modules can add dynamic content handling like additional features into Apache httpd. For example in order to handle PHP content, mod_php module can be loaded into Apache. Overall structure of Apache httpd operation is shown in Figure 3.4.

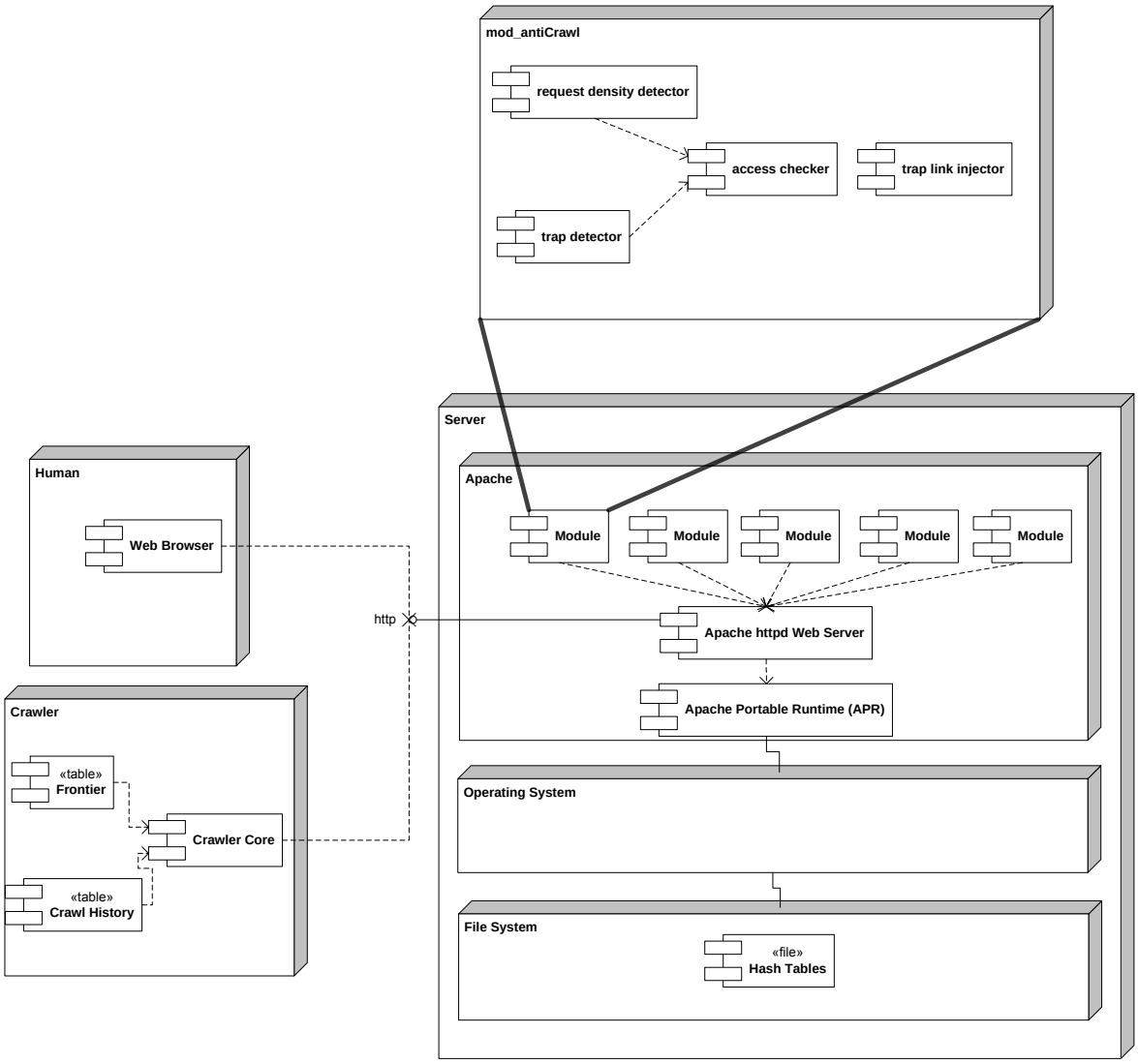

Figure 3.4. Overall Structure of Apache Server and mod_antiCrawl.

Apache modules are plug and play modules. Therefore, they start working when they are activated and stop working when they are deactivated. Module activation is also called module loading. Additionally, Apache httpd cannot communicate operating system directly. It communicates with the operating system via the middleware called Apache Portable Runtime (APR).

mod_antiCrawl is an Apache module implementation, which is the combination of Request Density Model and Hidden Link Model. mod_antiCrawl is implemented in C language by using Apache API [9]. UML activity diagram of mod_antiCrawl is given under APPENDIX B.3. Two hash tables are used as a data store in mod_antiCrawl

implementation because of the performance need. In order to handle hash table collision problem, a linked list structure is used within hash tables.

Inner steps of mod_antiCrawl can be explained as follows:

(i) When a request is received, the requester IP is searched in the Blacklist as a first step. If the IP exists in the Blacklist, server does not serves the page that is requested. In contrast, it sends an error page to client. This is the blocking functionality of the module.

(ii) If the IP is not in the Blacklist, requested URL is parsed in order to compare with the trap link. If the requested URL is trap link, the requester IP is added to the Blacklist and server sends an error page to the client. This is the crawler detection functionality of the module with hidden link catch.

(iii) If the requested page is not the trap URL, mod_antiCrawl checks if the IP is requesting too much in a short period of time. If the requester IP is sending too much requests, mod_antiCrawl assumes the client as a crawler. It adds IP address into the Blacklist and shows an error page as a response. This is the crawler detection functionality of the module with measuring request density.

(iv) If the request IP is not seen as a crawler, requested web page is served with an injected hidden trap URL in it. This is the injection functionality of the module.

Pseudo code of the access checking procedure of mod_antiCrawl is shown in Figure 3.5.

Since the module is planned to be plug and play, one of most the important implementations in this work is hidden link injection. It is thought that; leaving the hidden link injection to the developers and doing it statically is not an effective and intelligent solution. Because it decreases the usability of mod_antiCrawl. On the fly hidden link injection is mainly created in order to produce an effective and intelligent solution.

```
if IP is in Blacklist                                   //Check if IP is blocked
        if current time - punishment start time > blocking period
                DELETE IP from Blacklist
                WRITE IP to Timetable
                INJECT Hidden Trap Link to web page
                SHOW PAGE
        else
                SHOW ERROR PAGE
else                                                    //IP is not blocked
        if trap link is requested                       //Check if IP hits the trap
                WRITE IP to Blacklist
                DELETE IP from TimeTable
                SHOW ERROR PAGE
        else                                            //IP does not hit the trap
                if IP is in Timetable                   //Check if IP requested before
                                                        //Check if IP is sending too much
                        if current time - first arival time < time interval
                                if max count - 1 = counter
                                    WRITE IP to Blacklist
                                    DELETE IP from TimeTable
                                    SHOW ERROR PAGE
                                else
                                    counter++
                                    INJECT Hidden Trap Link to web page
                                    SHOW PAGE
                        else                            // IP is not sending too much
                                RESET counter
                                INJECT Hidden Trap Link to web page
                                SHOW PAGE
                else                                    //This is the first request from IP
                        WRITE IP to Timetable
                        INJECT Hidden Trap Link to web page
                        SHOW PAGE
```

Figure 3.5. Pseudo Code of the Access Checking Procedure of mod_antiCrawl.

In multiprocessing environments, more than one process might try to access or alter the same resource. However, this is a situation that must be solved in order to protect integrity and accuracy of the resource. mod_antiCrawl is a multi processing module as it is expected. Hash tables can be accessed by multiple processes at the same time. In order to organize processes and handle multiprocessing, mutex implementation is used in mod_antiCrawl. Details of the hash table and mutex implementations are explained below.

3.2.1. Hash Tables

Hash table is a data structure which uses a hash function in order to transform data to a hash value and the resulting hash value is used as an index of an array. In hash table implementations, modulus function is used after hash functions, in order to handle static structure of arrays. If the index value is larger than the size of the bucket, it is mapped to its modulus value within the size bound of the bucket. In order to achieve efficient results, bucket sizes are chosen as prime numbers and the modulus values are selected as the maximum prime number that is lower than bucket size.

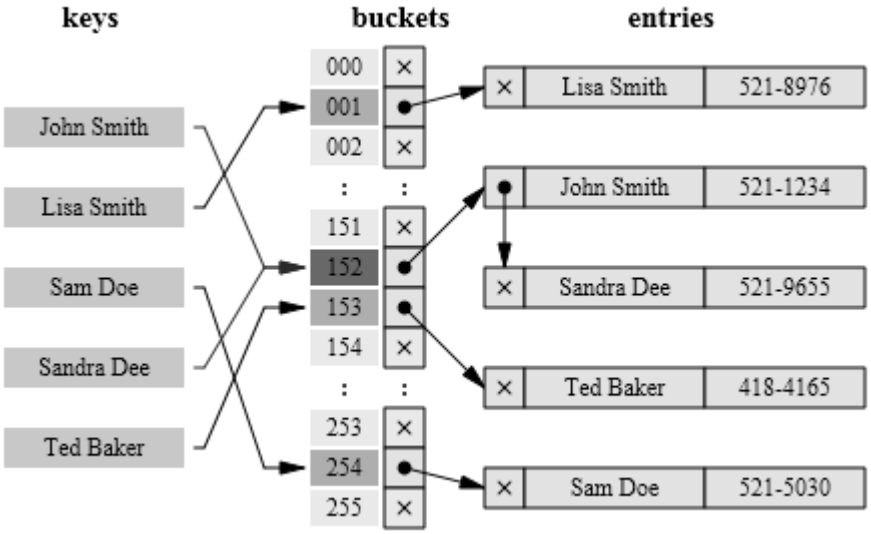

Figure 3.6. Example Hash Table With Separate Chaining [10].

Hash tables are expected to be uniformly distributed but in some cases, there might be collisions, means two different keys might have same indexes. Two different values might point to the same value after modulus function. To resolve collision problem, linked list is used as an array element. This approach is called Separate Chaining. Such hash tables are named with their data structures like "hash table of linked list". An example hash table with separate chaining is shown in Figure 3.6.

In a uniformly distributed hash table, operations like search, insertion and deletion can be carried out in a constant time value. Simply, operations are independent from the number of elements is stored in a hash table. In order to hold the Blacklist and the Timetable lists, hash tables are used in this study because of the performance advantage of

hash tables over other data structures. To solve collision problem and to achieve uniformly distributed hash table, size of table is chosen as a unique value and linked lists are used in each slot of the hash table.

In mod_antiCrawl implementation the Blacklist and the Timetable lists are merged and held in one hash table. In order to differentiate these two lists from each other, the Blacklist values are recorded into hash table with <IP>_BLOCKED pattern. An example hash table that contains both the Blacklist and the Timetable list values are shown in Figure 3.7. In this example, it can be seen that IP address 192.168.182.129 is in the Blacklist because of the BLOCKED suffix. It can be understood from the example hash table that, IP address 192.168.182.1 has made 5 requests before, and the first request is 1323879281 seconds later from the time 00:00 January 1 1970. This is the schematic of *time_t* values in C language and request time is recorded as *time_t* variable.

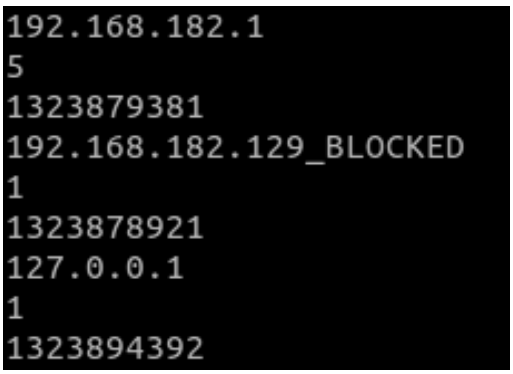

Figure 3.7. The View of the Blacklist and the Timetable in a Hash Table.

3.2.2. Mutexes

Mutex refers to mutual exclusion in concurrent programming that means exclusion of the shared resources from processes except currently using process. The shared memory usage problem was first identified by E. W. Dijkstra's paper [11]. In Dijkstra's paper, the problem is thought as more than one computers try to communicate with the same location and the solution was mentioned as these communications will take place one after other. In order to achieve this kind of exclusion, mutex has mainly two functions; mutex lock and mutex unlock. When a process uses a resource, it locks memory area that is treating and

prohibit other processes from using same memory area. At the end of the process, locked memory segments are released by using mutex unlock function.

In this work, hash table is stored in a shared file in a file system and it is protected from multi access conflict by mutex. Apache has to be multiprocessing module because of nature and as a result of multiprocessing, mutex must have used. However mutex might decrease the response time of the server. Because when more than one request is received at the same time - and it is very normal in web applications - one of the processes must lock the file that holds hash table, and other processes has to wait until the file lock is released.

4. PERFORMANCE EVALUATION

4.1. Performance Measures and Testing Environment

In evaluation phase, a test environment is set up and the performance of mod_antiCrlaw is evaluated. Given a specific number of web site files, we are interested in the number of files that are dynamically modified by mod_antiCrawl. Therefore the ratio of the number of existing files to the number of files discovered by crawlers is calcualted. Additionally, we are interested in how the web server response time is affected and load on the webserver is increased when mod_antiCrawl is activated.

A test environment is set up with a server machine that has 7.7GB memory and Intel Core2Quad Q9500 2.9 GHz processor. A test website which has about 2000 files (web pages, images, scripts, style sheets etc.) is used and different configurations of mod_antiCrawl are tested. The test setup of mod_antiCrawl performance evaluations is shown in Figure 4.1.

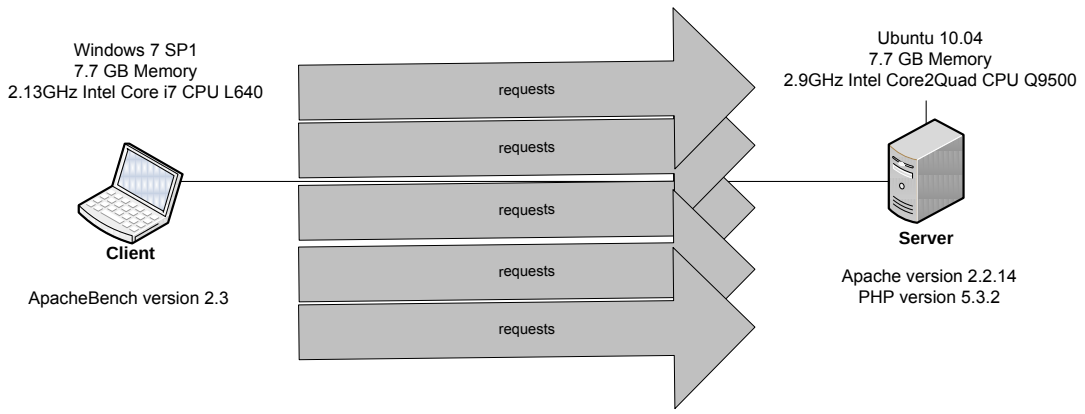

Figure 4.1. The Test Setup of mod_antiCrawl Performance Evaluations.

4.2. Crawler Discovery Tests

Crawler discovery tests are split into two parts. In the first part, only RDM configurations are set and HLM does not taken into account in order to observe RDM performance. Similarly in the second part only HLM functionalities are activated and

RDM does not taken into account to measure HLM performance. RDM tests are carried out in three different configurations as follows:

- Maximum 100 requests in 3 seconds
- Maximum 200 requests in 3 seconds
- Maximum 300 requests in 3 seconds

Maximum request numbers and time interval parameters of RDM are alterable. Thus, they are expected to be set by website administrators in real life. These values should be chosen in order to distinguish human users and crawlers. If the maximum request number value is chosen as small as human can perform, normal website users can be blocked by mod_antiCrawl assuming them as crawlers. For instance, "maximum 50 requests in 3 seconds" is a configuration that a human can perform. Also, if the values are chosen so large, mod_antiCrawl cannot detect crawlers, assuming them as normal website users. For example, "maximum 700 requests in 2 seconds" is a configuration that some crawlers can not perform. Consequently in evaluation phase, very small and very large values are not used.

The second part of crawler discovery tests is performed when only HLM configurations are set in two different configurations that are stated as follows:

- Inject trap link right before </html> closing tag
- Inject trap link right after closing tags

In performance evaluation phase of this work, three different crawlers are used in order to test mod_antiCrawl's crawler detection performance. Crawler functionalities of Acunetix and Netsparker web application vulnerability scanners are used. Additionally, a populer web crawler named Backstreet is used. Punishment duration is chosen as 3600 seconds, in other words 1 hour. Performance tests show that crawlers can find 1597 out of 2000 files in test website when mod_antiCrawl is passive. Crawlers could not find some files because they are not linked (referenced) in a page. When the configuration is set to "maximum 100 requests within 3 seconds", crawlers can find 290 files. In "maximum 200

requests within 3 seconds" configuration, finding number goes to 371 files and in "maximum 300 requests within 3 seconds" configuration crawlers could find up to 475 out of 2000 files. These values are average values for different crawlers.

In HLM tests, all the measurements are performed while Request Density Model is disabled as it is stated above. When trap link is injected before </html> tag, crawlers could find 245 files on average. When the hidden link inserted after anchor closing tag, mod_antiCrawl's performance increases and crawlers can only find averagely 153 files of the test website. All the measurements that are stated above are combined in a table and shown in Figure 4.2.

	Acunetix	Netsparker	Backstreet		
	Files			Average	Round Up
Module Deactive	1642	1588	1560	1596.67	**1597**
3-100	241	300	328	289.67	**290**
3-200	328	355	428	370.33	**371**
3-300	403	449	574	475.33	**476**
html	241	258	236	245	**245**
a	71	151	235	152.33	**153**

Figure 4.2. The Table of the Number of Files Found in Different Configurations.

According to these results,

(i) In the configuration of "hidden link before </html> closing tag", the hidden trap link is placed at the end of the page because </html> tag is the latest tag of a web page. But crawlers start to parse websites from the top and go to downwards. Therefore, crawler parses and extracts all the links that were placed above, until it reaches to hidden trap link. The problem can be solved by injecting traps after closing tag. Because when the trap is injected after tag, it can be placed in different areas of website more than one times. Every tag become followed by a trap link. As a consequence, it can be said that the location and the number of occurrences of the hidden link in a web page affects the performance of mod_antiCrawl. Anti-crawling module performs better when hidden link is occurred more than one times and placed in different areas of web page.

(ii) In Request Density Model configurations, mod_antiCrawl performs worse than Hidden Link Model configurations. Because it is very important to choose suitable values for counter and time interval parameters in Request Density Model. If the counter is chosen as a very small value, some normal users can be blocked and if it is chosen as a very large number, crawlers that run slowly, cannot be detected. Therefore, Hidden Link Model is an effective and easy to use feature of mod_antiCrawl and it is more suitable to the characteristics of web crawlers.

(iii) Even the worst performance of mod_antiCrawl, the number of files found are decreased about 70%. In hidden link configuration this ratio increases up to 90%. In Request Density Model, crawlers can find large number of files and this can be explained by its nature. Because the model has to permit requests until they reach up to a limit value in a predetermined time interval.

(iv) Hybrid (Hidden link after tag + 100 requests in 3 seconds) configuration does not perform better than "hidden link after tag" configuration. . Since, the crawler is detected by trap link controls, request density controls does not increase the performance of mod_antiCrawl.

All the measurements that are stated above are shown as a bar graph in Figure 4.3.

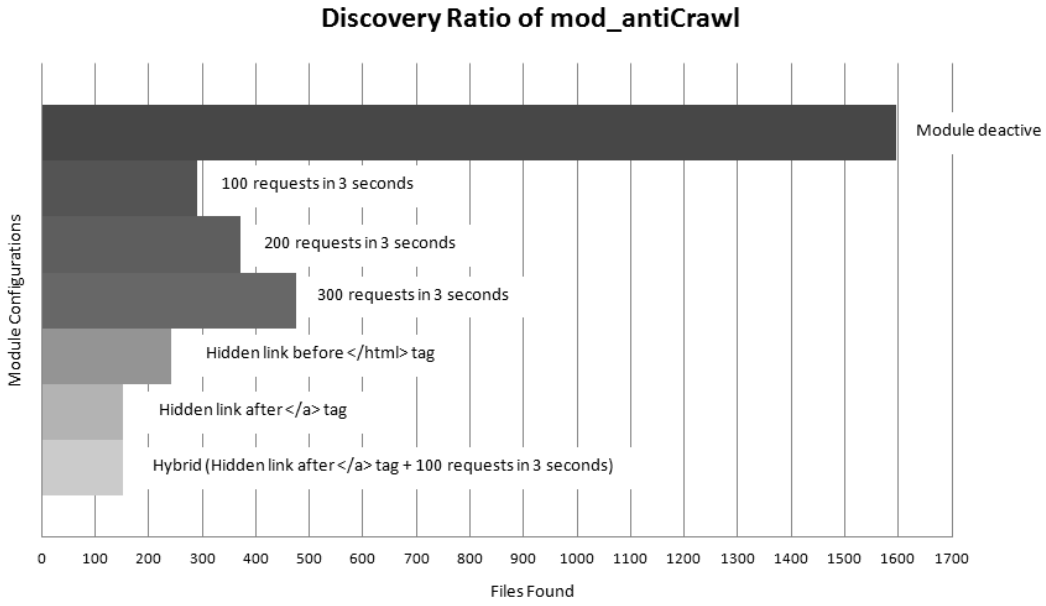

Figure 4.3. The Graph of the Number of Files Found in Different Configurations.

4.3. Server Load Tests

Besides the crawler detection performance of mod_antiCrawl, server overhead is also observed by measuring response time of the server and, memory and CPU usage of mod_antiCrawl. In server load tests, ApacheBench tool is used in order to send multiple and repetitive HTTP requests. ApacheBench, also called "ab" is an open source software that is used for benchmarking web servers. It has multiple and concurrent request sending capabilities. Response time measurements are done via Apache log files. Apache log level can be configured to show response time values in Apache acceess.log file.

Memory and CPU usage statistics are obtained from a different set up. In order to measure resource usage of mod_antiCrawl, SNMP (Simple Network Management Protocol) and Cacti integration is used. SNMP is a protocol that is designed to manage devices on IP networks. Different kinds of measurements can be performed in SNMP from temperature values to uptime values. Cacti is a solution that is used for representing SNMP measurements in a graphical form. In this thesis, SNMP and Cacti are installed on test server for measuring CPU and memory load.

According to the response time and load measurements of mod_antiCrawl in active and passive states with single source and multiple sources,

(i) In response time tests, a 40 KB sized HTML web page is used. Server can handle 1000 requests from a single source when mod_antiCrawl is disabled in average of 3425 microseconds for each. If the requests are received from 5 sources concurrently, response time value goes up to about 11913 microseconds These measurements can be seen in Figure 4.4. When mod_antiCrawl is activated, single source response time becomes 3686 microseconds for each requests and multiple sources value increases to 12011 microseconds in average which is shown in Figure 4.5. This negative effect is mainly the result of waiting mutex unlock for mutex locked shared resources while processing multiple requests.

(ii) In server load tests, ApacheBench tool is used to send 1000 HTTP requests in 5 concurrent threads repeatedly. When mod_antiCrawl is in passive state, memory

usage of server is about 1GB and 16% of CPU is used while handling responses These measurements are shown as graphs in Figure 4.6 and Figure 4.7. When mod_antiCrawl is activated, memory usage goes up to 4GB in average and CPU usage becomes about 19% as shown in Figure 4.8 and Figure 4.9. mod_antiCrawl holds hash table in a shared file that is accessed in each requests. Although there is a performance augmentation as an outcome of holding data on a hash table, file I/O processes increase the response time values in microseconds order after activating mod_antiCrawl. Since file I/O activities are slow when it is compared with memory I/O processes, load and CPU usage get higher.

Main focus of mod_antiCrawl development is to create an effective and complete crawler detection mechanism. Accordingly, performance of the module is not in the first priority of this work since some performance issues takes so much effort. As a result, these performance improvements like keeping hash tables in shared memory is planned to implement after crawler detection functionalities are completed.

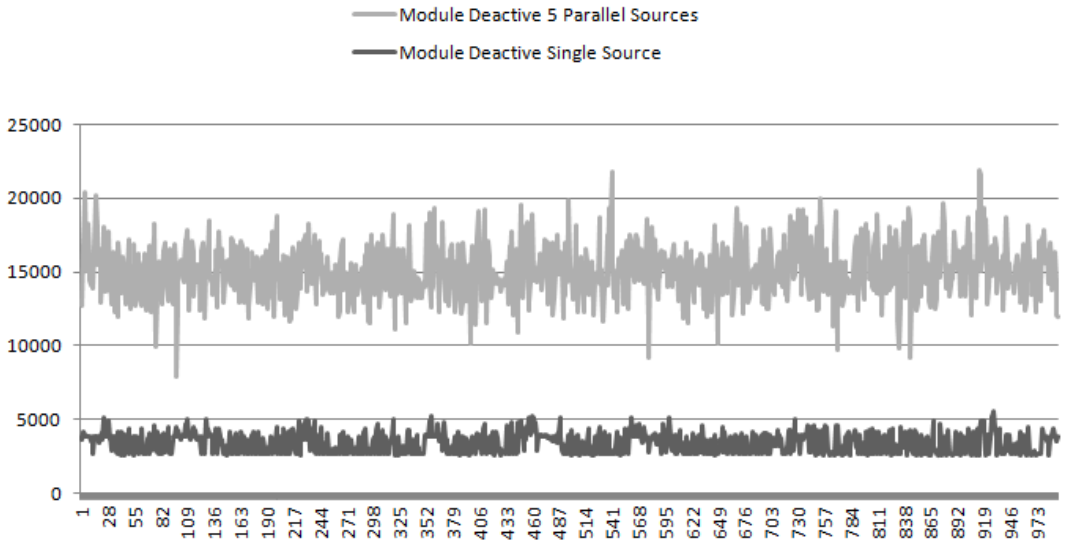

Figure 4.4. Response Time of The Apache Server When mod_antiCrawl is Inactive.

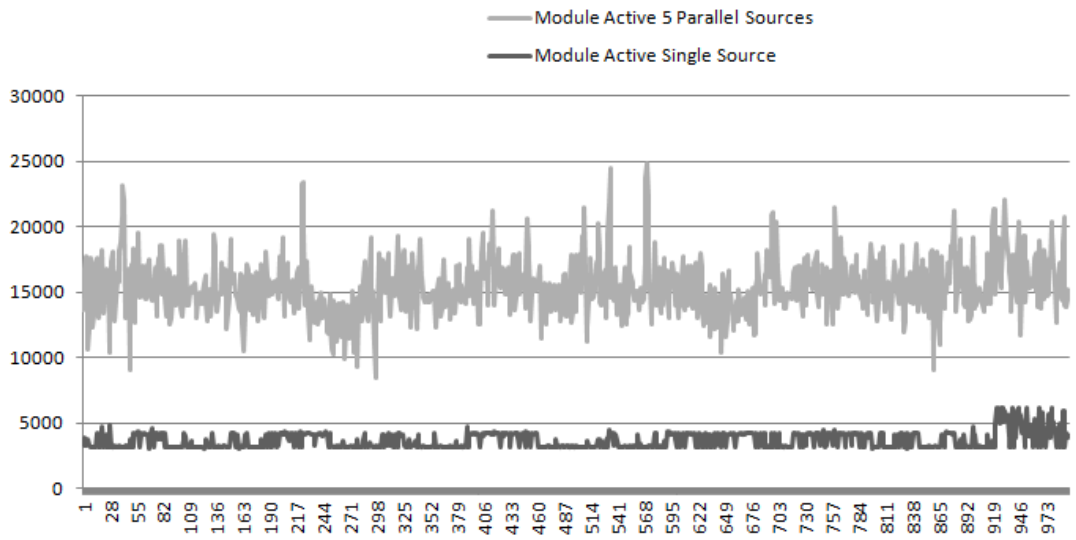

Figure 4.5. Response Time of The Apache Server When mod_antiCrawl is Active.

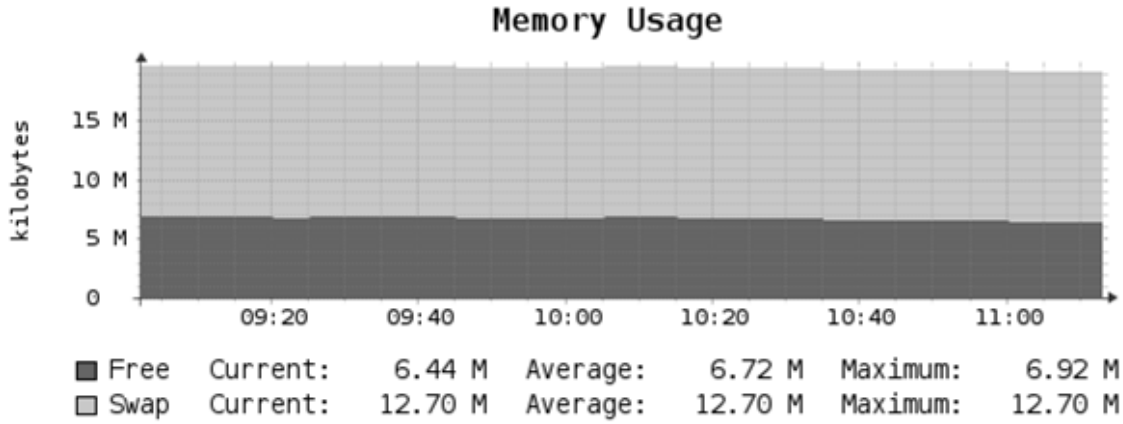

Figure 4.6. Free Memory of the Server When mod_antiCrawl is Deactivated.

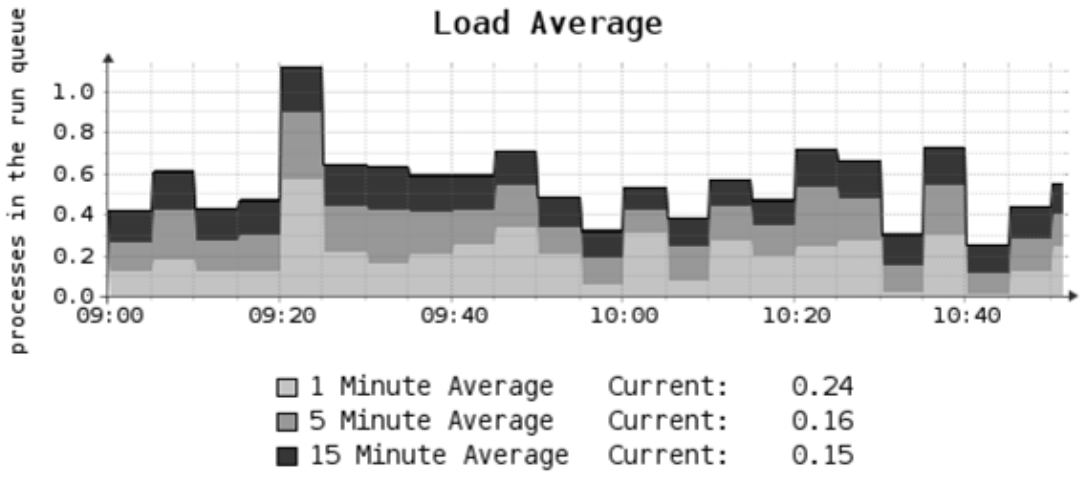

Figure 4.7. CPU Load of the Server When mod_antiCrawl is Deactivated.

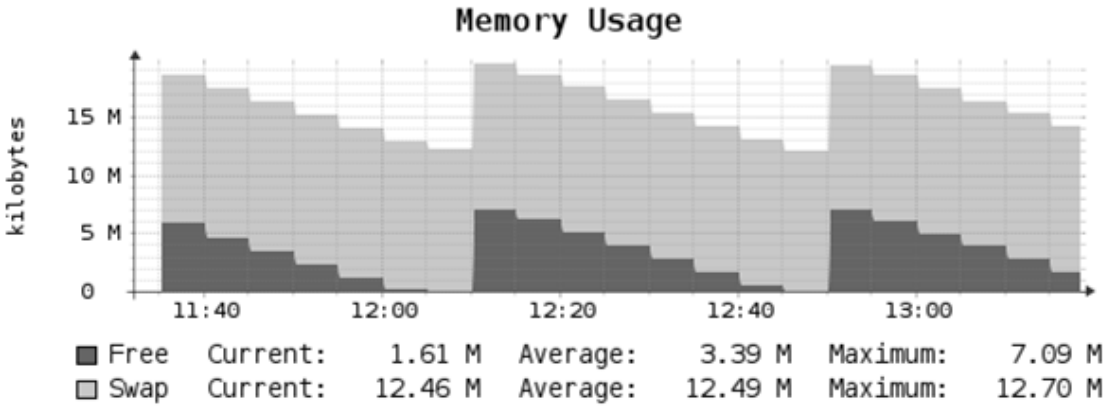

Figure 4.8. Free Memory of the Server When mod_antiCrawl is Activated.

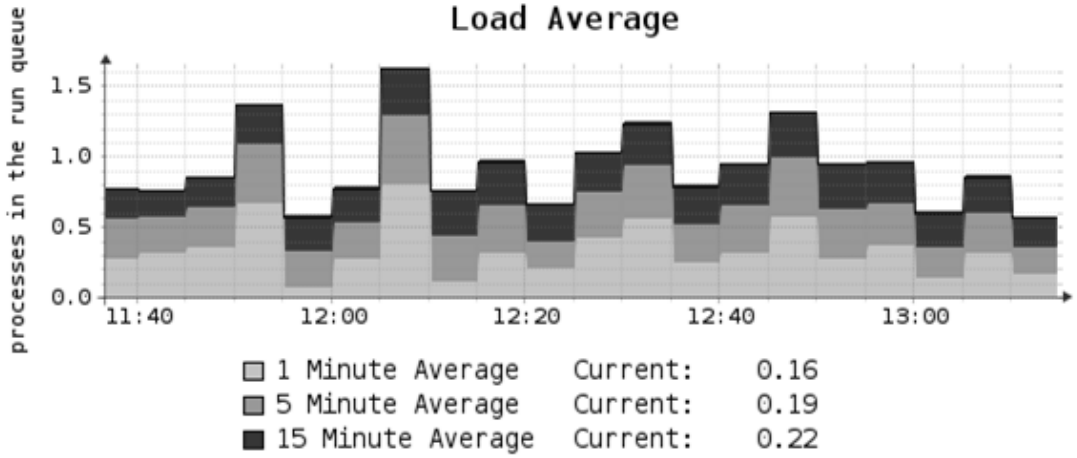

Figure 4.9. CPU Load of the Server When mod_antiCrawl is Activated.

In evaluation phase of this thesis, all the measurements are performed in a test environment. In order to bring simulated environment and real environment closer, a test website which is similar to live websites - contains more than 2000 files - is used in crawler findings test. Physical specifications like memory and CPU of the server machine are chosen similar to real servers. In load tests, thousands of requests are sent repeatedly for simulating crowded web servers.

5. CONCLUSION AND FUTURE WORK

Crawlers are beneficial and effective tools when they are aimed to increase search performance and accuracy. On the other hand, abuse of these crawling techniques is an issue that has to be protected personally and corporately. mod_antiCrawl is an Apache server extension which is developed in the context of this work that aims to serve crawler protections

In this thesis, a set of anti-crawling mechanisms are combined into an Apache web server module called mod_antiCrawl which has crawler detection and inhibition capabilities to protect servers from malicious crawlers. In order to provide an effective solution, behavioral characteristics of crawlers and precautions that can be taken are surveyed. mod_antiCrawl is developed in C language by using Apache API and hash tables are used in order to hold data. Hash tables are chosen because of the performance requirements of the module. As a solution for hash table collision problem, hash table with linked list sepeate chaining is used. Additionally, mutex is implemented to handle concurrent requests.

In performance evaluation phase, different kinds of tests are realized in order to observe mod_antiCrawl capabilities and measure the load on the system. Test steps are designed to simulate real environment. For this purpose, heavy network traffic is applied to see behavior of the system. According to the results, it is observed that crawler detection and inhibition skills of mod_antiCrawl are remarkable, despite it burdens to server. Even mod_antiCrawl decreases the crawler performances up to 90%, it increases the response time and the load of the server especially when IP lists and server are crowded. Therefore some module improvements and server hardenings can be realized to decrease resource usage of the system.

In order to increase performance and capabilities of mod_antiCrawl module, the following list identifies some future work:

(i) Injecting dynamically and randomly generated trap links instead of static trap links in order to obstruct more intelligent crawlers that can learn the trap links and do not hit them.

(ii) Evolving detaining features besides blocking features which are currently implemented. Detaining features does not stop crawling process, on the contrary they aim to preserve the crawling process forever. By developing these detaining features, mod_antiCrawl forces crawler into an infinite loop. As a result, crawling process is expected to be ineffective, unreliable and expensive.

(iii) Checking IP lists periodically, for eliminating some old records and increasing search performance. This step can be simply defined as resource management improvements.

(iv) Keeping hash tables in shared memory instead of file system in order to increase performance of mod_antiCrawl.

Current state of mod_antiCrawl is an important step for development of protection mechanisms against web crawlers that are not considered critical as well. The mod_antiCrawl project will be more beneficial for academic and open software community with the upgrades that are stated above.

APPENDIX A: mod_antiCrawl USER MANUAL

A.1. mod_antiCrawl Installation

The installation of mod_antiCrawl module is done as follows

mod_antiCrawl source code is compiled with *apxs* tool. *apxs* is the tool that is used for building and installing Apache extensions.

apxs -c mod_anticrawl.c

command compiles mod_anticrawl.c and generates three files in the same directory: mod_anticrawl.la, mod_anticrawl.lo and mod _anticrawl.slo

apxs -i mod_anticrawl.la

command installs module and adds mod_anticrawl.so shared object into apache modules directory. This directory contains dynamically installed Apache modules. As a next step of installation, mod_anticrawl.so must be loaded in Apache configuration file *httpd.conf* (or *apache.conf* in Debian based systems).

A.2. mod_antiCrawl Configuration

In order to use mod_antiCrawl, mod_anticrawl.so has to be loaded in configuration file. Loading is done by *LoadModule* directive in configuration file as shown below.

LoadModule anticrawl_module /usr/lib/apache2/modules/mod_anticrawl.so

LoadModule directive gets two attributes. First one is the name of the module that was declared in the source code and the second is the shared object file with full path.

Module parameters can be set in Apache configuration between *<IfModule></IfModule>* tags as follows,

```
<IfModule mod_anticrawl.c>
        HashTableSize 3097
        Count 100
        Interval 3
        BlockingPeriod 3600
        AddOutputFilterByType INJECT text/html
        Inject "s|</a>|</a><a href=dontclick.html style=display:none;>link</a>|ni"
</IfModule>
```

Figure A.1. Apache Module Configuration Example.

These HashTableSize, Count, Interval and BlockingPeriod values are configurable parameters of the module. If they are not set, default values are used by mod_antiCrawl. Their default values are set in mod_antiCrawl source code as HashTableSize is 3097, Count is 100, Interval is 3 and BlockingPeriod is 3600 where HashTableSize is the size of hash table that is used for holding Blacklist and Timetable, Count is the maximum request number in Interval time period, Interval is the time period in seconds that Count of requests are permitted and BlockingPeriod is the The Punishment period in seconds of detected crawlers

AddOutputFilterByType parameter adds the INJECT filter which is used for injecting hidden links. *Inject* parameter specifies which expression is replaced with hidden link suffix or prefix.

Inject "s||link|ni"

Expression defines that ** tag is replaced with

link

and these values can be altered in order to inject hidden link into another place.

These configurations can be combined into the expression that can be stated as follows,

```
LoadModule anticrawl_module /usr/lib/apache2/modules/mod_anticrawl.so
<IfModule mod_anticrawl.c>
        HashTableSize 3097
        Count 100
        Interval 3
        BlockingPeriod 3600
        AddOutputFilterByType INJECT text/html
        Inject "s|</a>|</a><a href=dontclick.html style=display:none;>link</a>|ni"
</IfModule>
```

Figure A.2. Apache Module Configuration.

APPENDIX B: MOD_ANTICRAWL DIAGRAMS

B.1. Request Density Model UML Activity Diagram

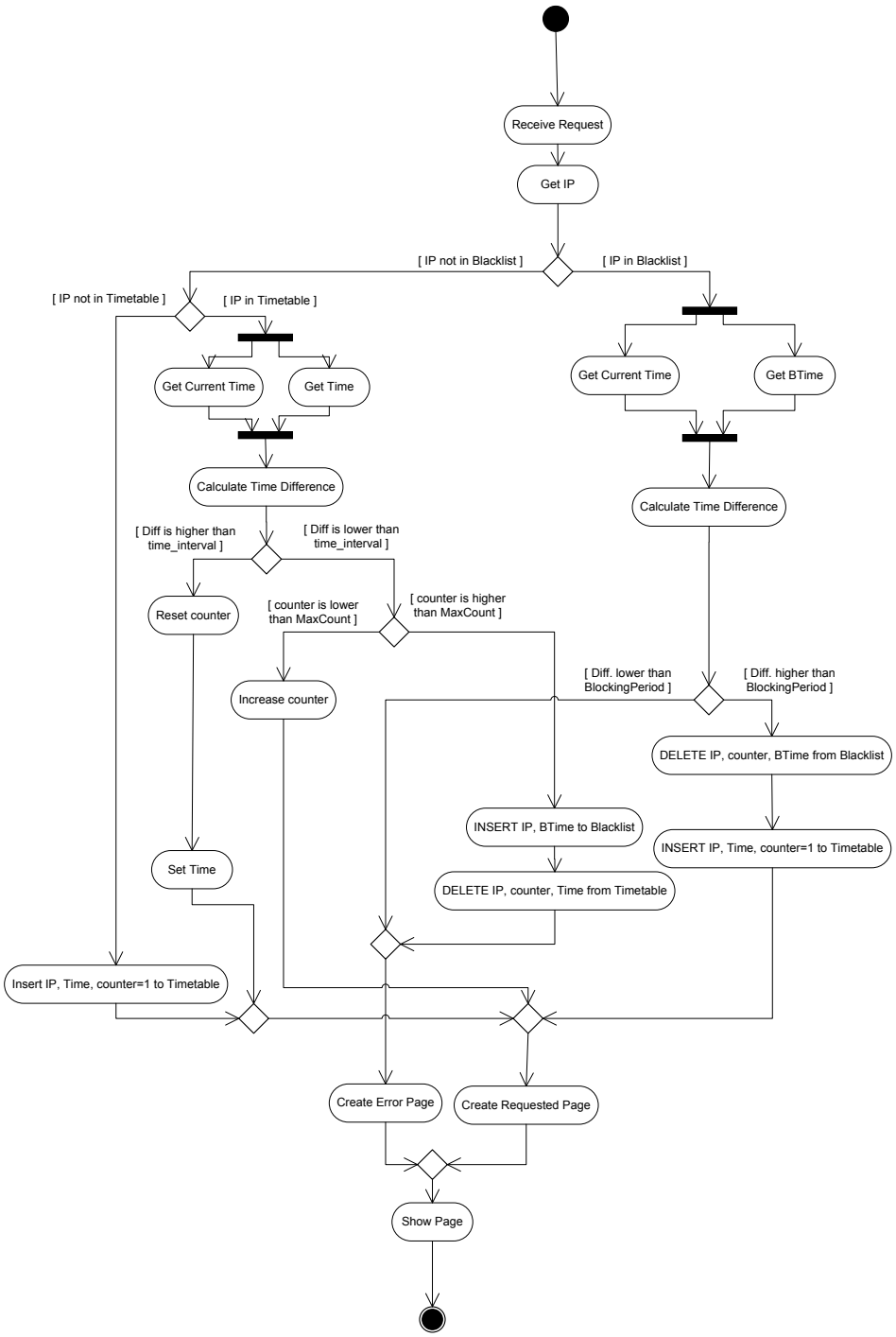

Figure B.1. RDM UML Activity Diagram.

B.2. Hidden Link Model UML Activity Diagram

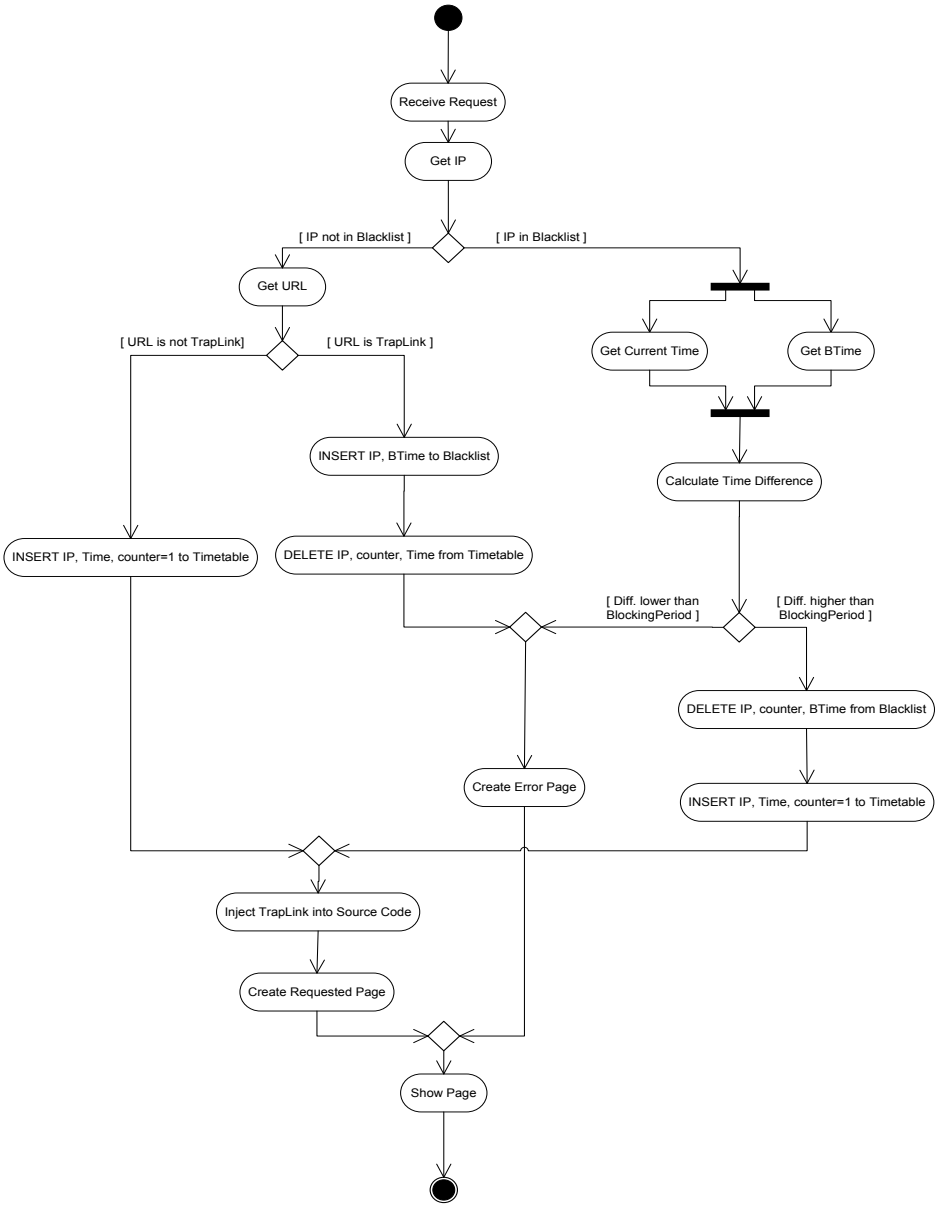

Figure B.2. HLM UML Activity Diagram.

B.3. mod_antiCrawl UML Activity Diagram

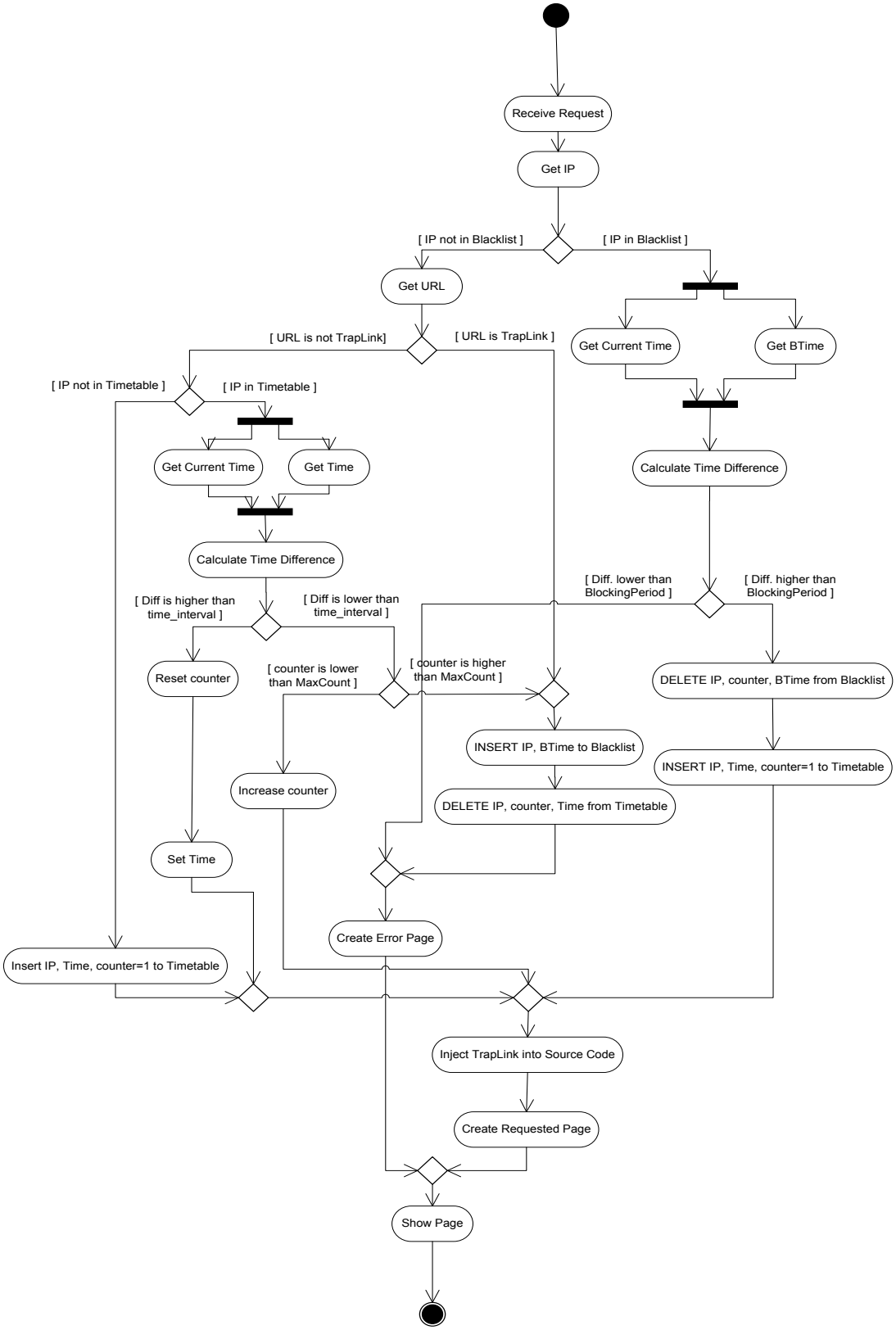

Figure B.3. mod_antiCrawl UML Activity Diagram.

REFERENCES

1. Apache, *Apache License*, 2004, http://www.apache.org/licenses/LICENSE-2.0.txt, accessed at 2012.
2. Netcraft, *Web Server Survey*, 2011, http://news.netcraft.com/archives/2011/12/09/december-2011-web-server-survey.html#more-5158, accessed at 2012.
3. Apache Software Foundation, *Third Party Apache Modules*, 2011, https://modules.apache.org/search.php?query=true, accessed at 2012.
4. Castillo C., *Effective Web Crawling*, Ph.D. Thesis, University of Chile, 2004.
5. Levene M., A.Poulovassilis, *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*, Springer, Berlin, 2004.
6. Tan P-N., V. Kumar, "Discovery of Web Robot Sessions Based on their Navigational Patterns", *Data Mining and Knowledge Discovery*, 2002.
7. Park, K., V. Pai, K.-W. Lee, and, S. Calo, “Securing Web Service By Automatic Robot Detection”, *In Proceedings of the 2006 Usenix Annual Technical Conference,* Boston, May 30–June 3, 2006.
8. Apc, *Anti Crawler*, 2009, http://apcanticrawler.sourceforge.net/, accessed at 2012.
9. Apache Software Foundation, *Developer Documentation for Apache 2.0*, 2011 http://httpd.apache.org/docs/2.0/developer/, accessed at 2012.
10. Wikipedia, *Hash Table*, 2012, http://en.wikipedia.org/wiki/Hash_table, accessed at 2012.
11. Dijkstra E.W., "Solution Of A Problem In Concurrent Programming Control", *Communications of the ACM*, Vol.8, No.9, 1965.

REFERENCES NOT CITED

Bau J., E. Bursztein, D. Gupta, J. Mitchell, 2010, "State of the Art: Automated Black-Box Web Application Vulnerability Testing", *IEEE Symposium on Security and Privacy,* California, 16-29 May.

Kew N., 2007, *The Apache Modules Book, Application Development with Apache*, Prentice Hall, New Jersey.

Kobayashi M., K. Takeda, 2000, "Information Retrieval On The Web", *ACM Computing Surveys*, Vol.32, No.2, 2000, pp.144–173.

Laurie B., P.Laurie, 2002, *Apache Definitive Guide,* O'REILLY, California.